Splinter MVVM Framework User Guide

Contents

[1 Document Control 3](#_Toc409431859)

[2 Glossary 3](#_Toc409431860)

[3 Introduction 3](#_Toc409431861)

[3.1 Who is this document for? 3](#_Toc409431862)

[3.2 What is MVVM? 3](#_Toc409431863)

[3.3 Prerequisites 4](#_Toc409431864)

[3.3.1 Software 4](#_Toc409431865)

[3.3.2 Knowledge 4](#_Toc409431866)

[4 Using the Framework 4](#_Toc409431867)

[4.1 Adding a reference 4](#_Toc409431868)

[4.2 Creating a View Model 4](#_Toc409431869)

[4.2.1 INotifyPropertyChanged Interface 4](#_Toc409431870)

[4.2.2 Inheriting the ViewModel Class 4](#_Toc409431871)

[4.2.3 Adding a View Model Property 5](#_Toc409431872)

[4.2.4 Raising PropertyChanged for Multiple Properties 5](#_Toc409431873)

[4.3 Binding to a View 6](#_Toc409431874)

[4.3.1 Using a View Model as Data Context 6](#_Toc409431875)

[4.3.2 Binding a Property to a UI Element 7](#_Toc409431876)

[4.3.3 Binding a Collection 8](#_Toc409431877)

[4.4 Sending a Message through the Broker 10](#_Toc409431878)

[4.4.1 The Publish – Subscribe Pattern 11](#_Toc409431879)

[4.4.2 Publishing a Message 11](#_Toc409431880)

[4.4.3 Subscribing to a Message Type 11](#_Toc409431881)

[4.4.4 Publishing a Message 13](#_Toc409431882)

[4.4.5 Customised Messages 13](#_Toc409431883)

[4.4.6 Advanced Broker Behaviour 14](#_Toc409431884)

[4.4.7 Unsubscribing 15](#_Toc409431885)

[4.5 Commands 15](#_Toc409431886)

[4.5.1 Creating a Command 15](#_Toc409431887)

[4.5.2 Command Parameters 16](#_Toc409431888)

[4.5.3 Binding a Command to a Button 16](#_Toc409431889)

[4.5.4 Using a Command to Publish a Message 17](#_Toc409431890)

[4.5.5 Overriding the CanExecute Method 18](#_Toc409431891)

[5 Framework Controls 18](#_Toc409431892)

[5.1 File Selection Control 19](#_Toc409431893)

[5.2 Nullable Check Box 19](#_Toc409431894)

[5.3 Animated Content Control 19](#_Toc409431895)

[5.4 Commandable Combo Box 20](#_Toc409431896)

# Document Control

|  |  |  |
| --- | --- | --- |
| Date | Information | Author |
| 27/02/2015 | Initial version complete. | Ian Hughes |
|  |  |  |
|  |  |  |
|  |  |  |

# Glossary

|  |  |
| --- | --- |
| Term | Description |
| WPF | Windows Presentation Foundation. A modern application user interface rendering subsystem. |
| MVVM | Model-View-View Model. An application architecture (see section 3.2). |
| XAML | Extensible Application Mark-up Language. A hierarchical language based on XML used to build user interfaces in WPF and Silverlight applications. |
| Binding | The process of associating the data presented in a UI window with data in an underlying class. |
|  |  |
|  |  |

# Introduction

## Who is this document for?

This document is for developers wishing to use the Splinter MVVM Framework to facilitate a WPF MVVM application. It discusses the use of Splinter, its classes, capabilities, and purpose. Using Splinter can allow developers to begin to develop powerful data-driven applications with an easy-to-use presentation logic flow.

## What is MVVM?

MVVM (Model-View-View Model) is a software architecture design pattern. It is a specific derivative of MVC (Model-View-Controller) for UI platforms that support event-driven programming.

MVVM dictates a separation of business logic from presentation logic. Splinter, the MVVM Framework discussed in this document, facilitates this by allowing UI code to remain in the View markup (using XAML in WPF) while logic behind the View is handled by a View Model class that allows easy data binding and automatic update of UI values. Commands allow binding of button clicks to behaviour and can further separate the View from the View Model. Additionally, Splinter includes a Broker class that uses a publish-subscribe model to allow View Models to publish events which other View Models may subscribe to. As you’ll quickly see, the Broker forms the heart of an application built using Splinter; it replaces the Messenger from MVVMLight and the EventAggregator from Prism, with the additional feature of allowing simple thread control of the messages.

## Prerequisites

### Software

* Visual Studio 2013 +

### Knowledge

* C# and .NET experience
* WPF experience

# Using Splinter

## Creating a View Model

In this section, we’ll create a simple View Model that we can later bind to a View.

### INotifyPropertyChanged Interface

Common in MVVM, the [INotifyPropertyChanged interface](http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifypropertychanged(v=vs.110).aspx) is used to provide a communication mechanism between a .NET class and a XAML view. The interface allows a developer to create a property that notifies any subscribing classes whenever it has changed. In WPF, this communication is created implicitly in XAML when a Binding is created. In a C# (or even VB) View Model class, we simply define a way to raise the PropertyChanged event when a property has changed. This means that whenever a property is changed, the UI is updated to reflect the change. In the Splinter MVVM Framework, a base class “ViewModel” has been created that already implements this functionality. There’s also a “SubscribingViewModel” class for View Models that subscribe to the Broker (see section 4.3) and unsubscribes when the View Model is disposing.

### Inheriting the ViewModel Class

The “ViewModel” class in the MVVM Framework already implements the INotifyPropertyChanged interface and provides a simple mechanism for allowing developers to quickly raise the PropertyChanged event on any changed property. To inherit the class, the Splinter.dll assembly must be referenced, and the Splinter namespace must be used. Then, you may inherit the class using the ‘:’ symbol, such as:

public class MyViewModel : ViewModel

{}

To make things easier, it is considered good practice as a naming convention to name any View Model class ending “ViewModel”. If you do the same with the View classes, it is much easier to see which View belongs to which View Model.

### Adding a View Model Property

Once a class has inherited from the “ViewModel” class, it’s important that any bindable properties – i.e. any property you wish to represent on the UI – are implemented in a specific way, in order that the PropertyChanged event is raised when needed. Note that a View Model is a class representing a View’s data, and therefore most properties are likely to be represented on the UI. If you find you have a number of non-displayed properties or fields, it is likely that you are breaking the MVVM pattern and using a View Model class for more than its purpose.

A property that will notify the UI when changed is likely to follow this pattern:

* A private field will be used to hold the value.
* A “get” accessor will allow public access to the field.
* A “set” accessor will use the ViewModel.SetPropertyAndNotify() method, taking a reference to the private field as well as the required new value.

An example of this type of field would look like this:

public class MyViewModel : ViewModel

{

private string \_name;

public string Name

{

get { return \_name; }

set { SetPropertyAndNotify(ref \_name, value); }

}

}

### Raising PropertyChanged for Multiple Properties

There are times when you might want to raise the PropertyChanged event for multiple properties when a single property changes, for example if you had a calculated property. Consider this property:

public bool NameIsNotBlank

{ get { return !String.IsNullOrWhitespace(Name); }}

Clearly this property does not have a “set” accessor, nor does it having a backing field. It is also clear that, should the UI depend upon this property, it will need to raise the PropertyChanged event whenever the Name property changes. To achieve this, we simply need to add a call to the ViewModel.OnPropertyChanged() method, passing a property name, in the “set” accessor for the Name property. Our View Model class will now look like so:

public class MyViewModel : ViewModel

{

private string \_name;

public string Name

{

get { return \_name; }

set {

SetPropertyAndNotify(ref \_name, value);

OnPropertyChanged(“NameIsNotBlank”);

}

}

public bool NameIsNotBlank

{ get { return !String.IsNullOrWhitespace(Name); }}

}

There are easier ways to achieve this using Splinter. The simplest is to use an override of ViewModel.SetPropertyAndNotify() that accepts any IEnumerable<string> for the names of the properties to notify a change for. That code might look like this:

public class MyViewModel : ViewModel

{

private string \_name;

public string Name

{

get { return \_name; }

set {

SetPropertyAndNotify(ref \_name, value, new[] {“NameIsNotBlank”} );

}

}

public bool NameIsNotBlank

{ get { return !String.IsNullOrWhitespace(Name); }}

}

If a single extra property is to notify of a change, we can pass the property directly into another overload.

public class MyViewModel : ViewModel

{

private string \_name;

public string Name

{

get { return \_name; }

set {

SetPropertyAndNotify(ref \_name, value, () => NameIsNotBlank );

}

}

public bool NameIsNotBlank

{ get { return !String.IsNullOrWhitespace(Name); }}

}

It’s also possible to use similar syntax to pass an IEnumerable of properties.

public class MyViewModel : ViewModel

{

private string \_name;

public string Name

{

get { return \_name; }

set {

SetPropertyAndNotify(ref \_name, value, new Expression<Func<object>>[] { () => NameIsNotBlank } );

}

}

public bool NameIsNotBlank

{ get { return !String.IsNullOrWhitespace(Name); }}

}

## Binding to a View

Now we have a simple View Model, we can start to create a View to represent it on the User Interface.

### Using a View Model as Data Context

A View class tends to be either a Window or User Control, depending on how it will be used. For this example, we’ll have a Window named MyView.

In order to use our View Model on the XAML View, we first need to reference it as an XML namespace. Use the “xmlns” XAML syntax to do so, and, since this is a .NET (CLR) namespace, we need to define it as such using “clr-namespace”. For the example, we are going to use the line

xmlns:viewModels="clr-namespace:MyApplication.ViewModels"

which we’ll place along with the other namespace declarations, in the <Window> tag.

Once we have a reference to the namespace in XAML, we need to declare an instance and, since we need to use XAML Binding, we should set it as the Data Context of the Window.

Within the Window tag, at the top for convention’s sake, use the following syntax to declare the instance and set it as Data Context:

<Window.DataContext>

<viewModels:MyViewModel/>

</Window.DataContext>

Note that Splinter also provides “DisposingWindow” and “DisposingUserControl” in the namespace Splinter.WpfControls, which implement IDisposable by calling Dispose() on the DataCOntext.

### Binding a Property to a UI Element

With our Window’s Data Context set as an instance of our “MyViewModel” (note that we have created an instance within the markup and without the use of the C# “new” keyword), we can create a control which is bound to the Name property of the View Model. Since that’s a string type that we’re going to want to edit, a TextBox seems the most appropriate choice. In this case, the binding is simple, and uses the following syntax:

<TextBox Text=”{Binding Name}”/>

The current MyView XAML should look like this:

<Window x:Class=MyApplication.Views.MyView

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:viewModels="clr-namespace:MyApplication.ViewModels">

<Window.DataContext>

<viewModels:MyViewModel/>

</Window.DataContext>

<Grid>

<TextBox Text=”{Binding Name}”/>

</Grid>

</Window>

If you set the MyView.xaml as the StartupUri in the App.xaml file and run the application, you’ll see a window with a text box, into which you can enter a field. You can’t see it yet, but any changes you make in the text box will change the Name property of the View Model. Let’s prove it, and prove our other View Model property (“NameIsNotBlank”) is also working.

Add a Button to the MyView class, using the following syntax:

<Button Grid.Row=”1” Content=”Say Hello”

IsEnabled=”{Binding NameIsNotBlank}”/>

Run the applicationagain, and you should see the button become enabled when any text is added to the text box, and disabled when cleared. Note you may need to change the TextBox’s Binding to “{Binding Name, UpdateSourceTrigger=PropertyChanged}” for the instant change to be acted upon, and you’ll need to add a second row to the existing Grid.

### Binding a Collection

Whenever a collection of objects needs to be presented on the screen, an ObservableCollection<T> should be used as a property of the View Model class (and is likely to contain a collection of classes inheriting from ViewModel itself). We’ll extend our current application to show how to do this. First, we’ll create a new View Model class named ItemViewModel.

public class ItemViewModel : ViewModel

{

private string \_displayText;

public string DisplayText

{

get { return \_displayText; }

set { SetPropertyAndNotify(ref \_displayText, value); }

}

}

We’ll extend the MyViewModel class to hold a collection of ItemViewModels.

public class MyViewModel : ViewModel

{

private string \_name;

public string Name

{

get { return \_name; }

set {

SetPropertyAndNotify(ref \_name, value);

OnPropertyChanged(“NameIsNotBlank”);

}

}

public bool NameIsNotBlank

{ get { return !String.IsNullOrWhitespace(Name); }}

public ObservableCollection<ItemViewModel> Items { get; set; }

public MyViewModel

{

Items = new ObservableCollection<ItemViewModel>();

Items.Add(new ItemViewModel { DisplayText = “foo” });

Items.Add(new ItemViewModel { DisplayText = “bar” });

Items.Add(new ItemViewModel { DisplayText = “baz” });

}

}

Notice we don’t need to do anything with the Items property, because an ObservableCollection already raises events when it changes, and a Binding will automatically update if set correctly. Update the MyView.xaml to the following:

<Window x:Class=MyApplication.Views.MyView

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:viewModels="clr-namespace:MyApplication.ViewModels">

<Window.DataContext>

<viewModels:MyViewModel/>

</Window.DataContext>

<Grid>

<TextBox Text=”{Binding Name, UpdateSourceTrigger=PropertyChanged}”/>

<Button Grid.Column=”1” Content=”Say Hello”

IsEnabled=”{Binding NameIsNotBlank}”/>

<ListView Grid.Row=”1” ItemsSource=”{Binding Items, Mode=OneWay}” DisplayMemberPath=”DisplayText”/>

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition Width=”Auto”/>

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition Height=”Auto”/>

</Grid.RowDefinitions>

</Grid>

</Window>

Run the application again, and you’ll see the 3 items from the collection represented in the ListView.

Now you have seen how to create a View Model and use it as Data Context for a View. You’ve seen how to bind properties to the UI, and how to create a responsive UI. Note that this introduction hasn’t covered all possibilities, as WPF’s Binding allows a lot of flexibility and control, but this should serve as an introduction.

## Sending a Message through the Broker

The Broker is an important part of the Splinter MVVM Framework; it is a singleton message handling class that enables easy, anonymous communication between types. In other words, it can allow a View Model class to respond to changes or actions handled in other View Models without either class needing a reference.

### The Publish – Subscribe Pattern

The Broker class is based around the “Publish-Subscribe” pattern. That means that a class can publish a message without knowing which classes (if any) are subscribed to it, and each subscribing class can respond to the message in a certain way.

### Publishing a Message

To use the broker, simply reference the Splinter.dll and use the Splinter.Messaging namespace within a class. This gives access to the singleton Broker instance (accessed using Broker.Current). The abstract Message class, which the Broker handles and all messages to be sent should be inherited from, is available in the namespace Splinter.Messaging.Messages.

### Subscribing to a Message Type

The Splinter.Messaging.Messages namespace includes the simple Message class in the library: TextMessage. The TextMessage class is a message with two properties: “Publisher”, which usually holds reference to the object that published the message (though bad practice might set this to null), and “Text”, a simple string.

If a class subscribes to the TextMessage type using the Broker, then that class will be notified whenever a TextMessage is published (unless the publisher targets a specific type or object) and will perform its action with it. It will then have access to the properties of the TextMessage, the Publisher and the Text.

To subscribe to a class, you can use the Broker.Subscribe<TMessage>() method, where TMethod is of the Message type. You must pass in a reference to the subscriber (using “this”) and the Action<TMessage> to be performed. Consider this class:

class Responder : IDisposable

{

public Responder()

{

Broker.Current.Subscribe<TextMessage>(this, OutputText);

}

private void OutputText(TextMessage msg)

{

Console.WriteLine(msg.Text);

}

public void Dispose()

{

Broker.Current.Unsubscribe(this);

}

}

The Responder class shown above does the following:

* Subscribes to messages of type TextMessage in its constructor.
* Implements IDisposable, and Unsubscribes itself from all messages on disposal (good practice).
* Has a method named OutputText which accepts a TextMessage as parameter, and this is the Action that is subscribed.

This means that, as long as a Responder instance has been created and not disposed, any time a TextMessage is published, the Text property will be output to the screen by that instance. If you had *x* instances of Responder, the Text property would be output *x* times.

Note that lambdas can be used for subscriptions, provided they fit the signature. Consider the benefit of the Responder written like this:

class Responder : IDisposable

{

public Responder()

{

Broker.Current.Subscribe<TextMessage>(this, msg => Console.WriteLine(msg.Text));

}

public void Dispose()

{

Broker.Current.Unsubscribe(this);

}

}

This is much briefer and, for simple message handling methods, can be preferable. However, avoid using lambdas for complex message handlers as this can be difficult to read, as well as reducing readability of the Stack while debugging.

Splinter.Messaging.Messages also includes an ObjectRelayMessage which allows the sending of any object.

### Publishing a Message

Publishing a Message is even more simple than Subscribing, since a Publish does not know which objects (if any) are going to take any action, and therefore no action is defined by the publishing object. Use the Broker.Publish<TMessage>() method to publish any Message type, noting that TMessage does not need to be defined since it can be implied from the Message being published.

Broker.Current.Publish(new TextMessage("A TextMessage has been published.", this));

That’s all that’s required. Now any subscribed objects will receive a TextMessage with a Publisher property that is a reference to the publishing object, and a Text property that holds the string “A TextMessage has been published.” If you had an instance of the Responder class created, it would output that text through Console.WriteLine().

### Customised Messages

Obviously, the Broker wouldn’t be very effective if all you could send were TextMessage instances. Any subscribers would respond to all the messages, and only strings could be sent. The good news is that developers can create their own messages for the Broker to work with by inheriting from the abstract Message class. This can then hold any number of properties, although it is recommended for performance purposes to try to keep the properties as few and as simple as possible, while being suitable for conveying the right information.

class NumberMessage : Message

{

public int Number { get; set; }

public IntMessage(int number, object publisher = null)

: base(publisher)

{

Number = number;

}

}

The NumberMessage class defined above is an example of how to inherit from the Message class, and is a simple message holding an integer property. Notice that the constructor allows the publishing object to be set, and sends the publisher to the base constructor. In this case, the default publisher is null.

### Advanced Broker Behaviour

The most common use of the Broker class has been defined, now it’s time to look at some of the more advanced uses of it.

#### Publish to Type

With a standard Publish operation (i.e. when the Broker.Publish() method is called, the message will be acted upon by all active subscribers. This might not be the behaviour you want for your application, and the Broker’s Publish() method has a few overloads allowing you to further control it. One example is the ability to Publish in such a way that only subscribers of a certain type will receive notification of the message. To do this, follow the below syntax:

Broker.Current.Publish(new TextMessage("Sent to type.", this), typeof(Responder));

Notice that there’s a new argument in the Publish() method, which accepts a Type. Any subscribing objects that are of the specified type will receive the message, while any other subscribing type will not be notified that it has been sent. In this way, developers are able to target only specific types when publishing messages.

#### Publish to Instance

It’s also possible to publish only to a specific instance using the Broker, provided you have reference to that instance. This is an edge case, since it’s likely that if a publishing object had reference to a subscribing object, it would be more likely to invoke its public methods then use the Broker.

var responder = new Responder();

Broker.Current.Publish(new TextMessage(“Sent to instance.”, this), responder);

This code demonstrates how to send to a specific instance. In this code, only the Responder instance named “responder” will receive the TextMessage, even if there are other subscribers to the TextMessage type.

#### Controlling the Thread

Previous code has all shown messages being published and handled to on whichever thread they have been published from. The Broker provides additional functionality to control the type of thread being used to both publish and respond to messages as they are sent. A publisher might choose to send all messages on a new thread, meaning all subscribers’ actions will occur on that new thread. However, any subscribers might also choose to handle the message on the UI thread (the WPF Dispatcher thread) to allow UI elements to be changed.

Thread decisions are handled by selecting from the “ThreadOption” enum. There are three options:

* “Current” – the current thread will continue to be used. This is the default option when publishing and handling messages.
* “New” – a new thread will be created by the .NET ThreadPool to publish and handle the messages.
* “Dispatcher” – the Broker will attempt to use the WPF application’s Dispatcher to publish or handle the message. If the WPF Dispatcher cannot be used, for example if the code is invoked from a non-WPF application, then the thread option will default to Current.

To control the thread used when publishing, use the following syntax:

Broker.Current.Publish(new TextMessage("Sent on new thread."), ThreadOption.New);

To control a subscriber’s thread option, use the following syntax:

Broker.Current.Subscribe<TextMessage>(this, o => Console.WriteLine(o.Text), ThreadOption.Current);

### Unsubscribing

In the Responder class, we had added a Dispose() method which unsubscribed the Responder instance from all message types. To unsubscribe from all message, simply call Unsubscribe passing reference to the object. From the instance itself, you would use

Broker.Current.Unsubscribe(this);

It’s also possible to remove only subscriptions to a specific message type, by using a TMessage generic argument. The syntax for this is

Broker.Current.Unsubscribe<TextMessage>(this);

To go even deeper than that, if your subscriber had multiple message handlers that are subscribed to the Broker’s queue and you want to remove just one handler, you can do that too, by specific the message handling Action as well as the TMessage type parameter. The syntax is

Broker.Current.Unsubscribe<TextMessage>(this, OutputText);

## Commands

So far we’ve covered how to create a simple View Model class and bind its properties to a View using the powerful XAML features. However, few user interface applications will be limited to this simplicity, and will often include UI features such as buttons. When a button is pressed, the user expects something to happen. The Splinter MVVM Framework includes a way to write that something: the Command class.

### Creating a Command

The Command class is another abstract class in the Framework, and is in the Splinter.dll within the namespace Splinter (along with the ViewModel class). To create a command that can be bound to a WPF control, inherit the Command class.

class SayHelloCommand : Command

{

public override void Execute(object parameter)

{

MessageBox.Show(“Hello.”);

}

}

A class that inherits from Command must override the Execute() method. This method defines the code that executes when the command is invoked, for example when the user clicks the button the command is bound to. The method also takes a single parameter, which is of type System.Object, and this will be covered in the next section.

### Command Parameters

Often, you’ll need to feed some kind of information to a command before doing anything, and that is what the command parameter is for. In the Command class, this will be fed through to the Execute() method, and comes from the UI as a Dependency Property of the Control the Command is bound to. By default, the parameter will be null if not set on the control. In other situations, the parameter might be any kind of object, whether a View Model, one of its properties, or something from the view itself. Just as with a Message, it’s good practice to pass as simple an object as you can through the Command Parameter, while ensuring all necessary information is provided.

For the SayHelloCommand, we’ll want to pass a name to say hello to (in fact, we’ll pass the string Name property from the MyViewModel in the earlier example, and bind the SayHelloCommand to the button in MyView).

With that in mind, the best course of action is to cast the “parameter” to a string before performing the necessary operations. It’s often a good idea to perform a safe cast, and return if the result is null.

public override void Execute(object parameter)

{

var nameString = parameter as string;

if(nameString == null) return;

MessageBox.Show(

String.Format(“Hello, {0}.”, nameString));

}

### Binding a Command to a Button

In order that a button can invoke a Command, a binding needs to take place. To bind the SayHelloCommand to the Button in the MyView, you could start by creating an instance of the SayHelloCommand to use (I would add this to the Window.Resources section). Note that I have added the commands namespace to the file using “xmlns”, giving it the alias “cmd”.

<Window.Resources>

<cmd:SayHelloCommand x:Key=”SayHelloCommand”/>

</Window.Resources>

This syntax has created an instance of SayHelloCommand that is accessible to our Window and any of its children, accessible using its key “SayHelloCommand” as a resource. Now, to bind the command and the MyViewModel’s Name property as a parameter, change the button like so:

<Button Grid.Column=”1” Content=”Say Hello”

IsEnabled=”{Binding NameIsNotBlank}”

Command=”{StaticResource SayHelloCommand}”

CommandParameter=”{Binding Name}”/>

Run the application, enter a name and click the button. The application should now greet you by name thanks to the binding.

### Using a Command to Publish a Message

Because of the detached nature of MVVM and the Splinter Framework, it’s likely that often a Command class will be used to publish a Message through the Broker. This allows an event, such as a button click, to indirectly notify other View Model classes of the event, and even pass data easily between View Models. We’re going to take a look at how to do this using what we’ve learned already.

First, we’re going to change the SayHelloCommand class, so that it publishes to the Broker instead of simply creating a MessageBox. The code for the SayHelloCommand’s Execute method will now look like the below:

public override void Execute(object parameter)

{

var nameString = parameter as string;

if(nameString == null) return;

Broker.Current.Publish(new TextMessage(nameString, this));

}

Running the application now and the button won’t appear to do anything; it does, it publishes a message. However, there is currently no subscriber for the Broker to notify. Add the following to the MyViewModel’s constructor:

Broker.Current.Subscribe(this, msg => MessageBox.Show(

String.Format(“Hello, {0}.”, msg.Text)));

We have moved the handling of the event to the View Model to demonstrate the use of the Broker from a Command. Note that, of course, any Message type can be published, allowing much more complicated Event -> Publish -> Subscriber Action models to be built.

### Overriding the CanExecute Method

The Command class, which is abstract, has a structure for handling whether it can be executed. If a Command is bound to a Button control and its CanExecute returns false, the button will show disabled. By default, the Command class’ CanExecute will return true. However, it can also be set when needed by using the following syntax:

ChangeCanExecute(false);

The Boolean value provided to this method will set the return of the CanExecute to that same value; therefore, you can control whether a Command is executable in code this way.

# Framework Controls

The Splinter MVVM Framework comes with some WPF Controls to extend the WPF control palette. Note that Splinter should be compatible with any WPF controls (including those in the WPF Toolkit) since it uses Dependency Property for binding. The Control library could and should be extended as new customised, generic controls are created. The controls are available in the Splinter.dll library file.

## File Selection Control

WPF does not feature a standard control for file selection, and therefore this has been added to the Framework. The FileSelectionControl features a TextBox which the file’s path is bound to (the FilePath property), as well as an ellipsis “…” button allowing the user to browse with a File Dialog.

The control can be used for both file open and file save operations. To set it as a save control, set the IsSaveFile property to true.

You can optionally provide a file filter to prevent opening or saving specific file types. This uses the same format as standard Windows dialog filters, and should be set to the FileFilter property.

There are some other properties that might be used. InitialDirectory allows you to set the directory in which the Dialog will open (this can be a bound property too, so it can be changed at runtime). CheckFileExists will, if set to true, check the file exists before completing the Dialog use. The OverwritePrompt will, if IsSaveFile is set to true and the selected file exists, will prompt the user to overwrite the file.

Below is an example in XAML of this control.

<fileAccess:FileSelectionControl FilePath="C:/File.txt" IsSaveFile="True" InitialDirectory="C:/" OverwritePrompt="True"/>

## Nullable Check Box

The standard WPF CheckBox has only 2 states; true and false. Only a Boolean can be assigned to the IsChecked value. The NullableCheckBox control extends this slightly, allowing use of a nullable Boolean (bool? or Nullable<bool>) and indicating it appropriately.

## Animated Content Control

The AnimatedContentControl is a visual control that allows use of any selections of UserControls (Views, in this case) and allows the developer to control the animation that occurs when transitioning between them. I recommend testing and playing with this control to see what results you can get.

To set the displayed content, bind a ViewModel class with a template View to the Content property of the AnimatedContentControl. Then, when the Content changes, the transition animation will occur.

The MovementDirection property allows you to set which way a control will move off and on screen. Options are Up, Down, Left and Right. The default is Left.

The EasingMode property defines what type of easing to apply to the animation. EasingMode is of type [System.Windows.Media.Animation.EasingMode](http://msdn.microsoft.com/en-us/library/system.windows.media.animation.easingmode%28v=vs.110%29.aspx) and options are EaseIn, EaseOut and EaseInOut.

The EasingFunctionType property allows the developer to control the Easing Function applied to the animation: this is how the AnimatedContentControl really opens up. Options are BackEase, BounceEase, CircleEase, CubicEase, ElasticEase, ExponentialEase, PowerEase, QuadraticEase, QuarticEase, QuinticEase and SineEase. You may further control these by using the AnimatedContentControl’s EasingFunctionOptions property, which is of type EasingFunctionOptions (a class that has an inherited type for each Easing function defined above).

Finally, you may control how long the transition animation will take (in seconds) with the AnimationSeconds property.

An example of the AnimatedContentControl in XAML is below:

<cntrl:AnimatedContentControl Content="{Binding CurrentViewModel}"

MovementDirection="Left" EasingFunctionType="BounceEase"

EasingMode="EaseInOut" AnimationSeconds="0.5">

<cntrl:AnimatedContentControl.EasingFunctionOptions>

<easingFunctionOptions:BounceEaseOptions Bounces="3"

Bounciness="10"/>

</cntrl:AnimatedContentControl.EasingFunctionOptions>

</cntrl:AnimatedContentControl>

## Commandable Combo Box

The standard WPF ComboBox does not invoke a Command when the selection is changed; the CommandableComboBox does. It’s that simple: simply bind a Command and CommandParameter just like you would a Button. Since the CommandableComboBox inherits from ComboBox, it keeps all the remaining features.